UNIVERSIDADE FEDERAL DE UBERLÂNDIA

Thiago Pereira Muniz

Estudo comparativo do uso de gerenciadores de bancos de dados relacionais e não relacionais para a manipulação de documentos JSON

Uberlândia, Brasil

2023

UNIVERSIDADE FEDERAL DE UBERLÂNDIA

Thiago Pereira Muniz

Estudo comparativo do uso de gerenciadores de bancos de dados relacionais e não relacionais para a manipulação de documentos JSON

Trabalho de conclusão de curso apresentado à Faculdade de Computação da Universidade Federal de Uberlândia, como parte dos requisitos exigidos para a obtenção título de Bacharel em Sistemas de Informação.

Orientador: Profa. Dra. Maria Camila Nardini Barioni

Universidade Federal de Uberlândia – UFU

Faculdade de Computação

Bacharelado em Sistemas de Informação

Uberlândia, Brasil

2023

Thiago Pereira Muniz

Estudo comparativo do uso de gerenciadores de bancos de dados relacionais e não relacionais para a manipulação de documentos JSON

Trabalho de conclusão de curso apresentado à Faculdade de Computação da Universidade Federal de Uberlândia, como parte dos requisitos exigidos para a obtenção título de Bacharel em Sistemas de Informação.

Trabalho aprovado. Uberlândia, Brasil, 01 de novembro de 2023:

Profa. Dra. Maria Camila Nardini Barioni
Orientadora

Professor

Professor

Uberlândia, Brasil
2023

Lista de ilustrações

Lista de tabelas

Sumário

1 Introdução

O crescente volume, apresentação e padronização de dados das tecnologias envolvidas na Quarta Revolução Industrial têm aberto novas oportunidades para a humanidade estudar ou resolver problemas antes muito complexos (FREDERICK, 2016). O aumento do poder de processamento, miniaturizado e por menores custos, tem levado à proliferação de dispositivos e sistemas capazes de produzir e armazenar enormes quantidades de dados. No mundo da tecnologia e na sociedade, os modelos de negócios têm aumentado gradativamente, fazendo com que novos sistemas e processos sejam criados frequentemente para atendê-los (FOOTE, 2018).

Um dos pontos de decisões na construção de um sistema que lide com tais dados é a escolha do modelo de dados apropriado para a criação do banco de dados que os armazenarão e posteriormente recuperarão as informações salvas. A escolha incorreta de tal banco de dados ou seu tipo pode acarretar em um sistema de difícil manutenção e que poderá rapidamente ficar depreciado, entrar em desuso e finalmente ser descartado.

Dentre todas as categorias de sistemas gerenciadores de bancos de dados, as principais e mais usadas são os baseados no modelo Relacional e nos modelos NoSQL(*Not Only SQL*). O modelo relacional é um dos mais antigos, guardando informações em tabelas, com colunas e linhas (tuplas de colunas). O modelo de dados relacional foi proposto pela IBM na década de 70, que então tornou-se o modelo padrão de banco de dados para aplicações comerciais e se mantém relevante e muito utilizado nos dias atuais (UYANGA et al., 2021).

Com o passar dos anos e da latente necessidade de integração com informações dispostas no formato JSON(PHIRI; KUNDA, 2017), sistemas gerenciadores de bancos de dados relacionais passaram a implementar o tipo de dado JSON de forma nativa em sua base, como por exemplo o MySQL, da Oracle. O MySQL é um sistema de gerenciamento de banco de dados relacional que utiliza SQL como linguagem de interface e possui em sua implementação a possibilidade do uso do tipo de dado nativo JSON, definida pela RFC 7159. Ele consegue validar um documento JSON automaticamente e tem sua estrutura otimizada para lidar com rápidos acessos de leitura a estes documentos (ORACLE, 2022).

Os bancos de dados NoSQL são mais recentes, nasceram da necessidade de processamento de dados não estruturados, em maior volume, com mais velocidade e flexibilidade para sistemas *web*. Não possui apenas um sistema de persistência de dados, mas pode ser dividido em outras quatro categorias principais dependendo da implementação de seu modelo de dados: Chave-Valor, Documentos, Família de Colunas ou Grafos(SADALAGE; FOWLER, 2012).

Dentre os modelos de dados NoSQL, o modelo baseado em documentos permite a manipulação de dados não estruturados de forma flexível, possibilitando a adição e remoção de campos sem a necessidade de modificar a estrutura global dos documentos. Um dos bancos de dados mais usados que adota esse modelo é o MongoDB(DB-ENGINES, 2022). O MongoDB é um sistema de gerenciamento de banco de dados NoSQL orientado a documentos, desenvolvido pela MongoDB Inc., que permite o armazenamento e a recuperação eficiente de grandes volumes de dados no formato de documentos, oferecendo alta escalabilidade e desempenho(MONGODB, 2023).

No trabalho de conclusão de curso descrito aqui, foi explorada a utilização do MongoDB como uma alternativa ao modelo relacional, representado pelo MySQL. Eles foram usados neste estudo para comparar diversas operações de inserção e manuseio de grandes volumes de dados no formato de documentos JSON.

1.1 Objetivos

O presente trabalho tem como objetivo realizar uma análise comparativa de dois bancos de dados em diferentes situações de uso, por meio da interação com grande volume de dados dispostos em formato JSON. Com a realização de testes de execução de tarefas, pretende-se demonstrar como cada modelo apresenta vantagens em relação ao outro, considerando os parâmetros comparativos utilizados no mercado. Entre esses parâmetros, destacamos a velocidade necessária para persistir uma quantidade finita de dados e a recuperação de um volume finito de registros com tamanhos específicos, além de todos os procedimentos necessários para atingir essas finalidades.

Com base nisso, é possível concluir sobre algumas diferenças entre os dois gerenciadores de bancos de dados. Essa análise permitirá uma compreensão aprofundada do desempenho de cada modelo e, consequentemente, facilitará a tomada de decisões mais assertivas em relação à escolha do melhor banco de dados para cada caso específico.

1.2 Justificativa

Este trabalho de pesquisa pode fornecer informações valiosas para desenvolvedores e engenheiros de software em projetos que lidem com a necessidade de escolher o melhor banco de dados para armazenamento de grandes quantidades de dados no formato de documentos JSON. Além disso, esse estudo visa esclarecer dúvidas e desmistificar possíveis concepções equivocadas relacionadas à manipulação desses dados por diferentes tipos de gerenciadores de bancos de dados.

Essa pesquisa pode ajudar a tomar decisões informadas sobre qual gerenciador de banco de dados usar em diferentes situações, levando em consideração fatores como

desempenho na execução de tarefas nesses bancos de dados, facilidade de uso e custos ao comparar o MySQL com a implementação do tipo JSON ao MongoDB. Toda a documentação do processo, desenvolvimento e uso das ferramentas para a execução das tarefas contida neste trabalho também tem como finalidade auxiliar futuras pesquisas de natureza comparativa. Com pequenas alterações, deve ser possível realizar diversas tarefas com outros bancos de dados e até mesmo diferentes fontes de dados para as mesmas.

1.3 Organização do Texto

Este trabalho está organizado em cinco capítulos que abordam de forma estruturada os elementos fundamentais do trabalho realizado. No capítulo 2, Referencial Teórico, são apresentados os conceitos básicos relacionados aos dois modelos de banco de dados amplamente utilizados: o banco de dados relacional e o NoSQL. No seguinte capítulo, Trabalhos Correlatos, são expostos diversos trabalhos de pesquisa que abordam temas semelhantes ao tratado neste trabalho. Esses estudos incluem pesquisas que discutem a evolução, aplicabilidade e comparação entre bancos de dados NoSQL e relacionais, fornecendo uma visão abrangente sobre o assunto. No capítulo Método de Trabalho são descritos os procedimentos adotados na pesquisa, que abrangem uma abordagem quali-quantitativa para compreender o comportamento dos bancos de dados. São apresentados os detalhes sobre a ambientação, obtenção dos dados e os códigos de programação utilizados para a realização das tarefas estipuladas nos bancos de dados. Em seguida, no quinto capítulo, é realizada uma análise detalhada dos dados gerados durante a execução das tarefas nos bancos de dados. São apresentados os tratamentos e cálculos realizados, bem como a interpretação dos resultados obtidos. As tabelas de resultados exibem os valores de tempo médio em segundos, obtidos através do console ao final de cada script tarefas executada. Por fim, o último capítulo conclui o trabalho, destacando e considerando os resultados e além disso, são fornecidas recomendações para a continuidade do trabalho, com sugestões de possíveis direções futuras para pesquisa e desenvolvimento na área de banco de dados em geral.

2 Referencial Teórico

O armazenamento e gerenciamento de dados são essenciais para a grande maioria das aplicações. E o modelo de banco de dados é um fator crucial para a eficiência e desempenho dessas aplicações. Nesse sentido, dois modelos se destacam: o banco de dados relacional e o NoSQL. Ambos apresentam vantagens e desvantagens que devem ser consideradas no momento da escolha. Por isso, neste capítulo de referencial teórico, foram introduzidos conceitos básicos sobre esses dois modelos de banco de dados, proporcionando um alicerce para o entendimento deste trabalho que lida com diferentes gerenciadores de banco de dados e execuções de tarefas nos mesmos.

2.1 Banco de dados Relacional

O modelo relacional de dados é amplamente utilizado em sistemas de bancos de dados, uma vez que fornece uma maneira organizada e estruturada de armazenar informações. Segundo Date (2004), o modelo é composto por três aspectos principais: estrutural, integridade e manipulação.

No aspecto estrutural, os dados são organizados em tabelas, onde cada tabela representa uma entidade ou objeto do mundo real. Cada linha da tabela representa uma ocorrência desse objeto, e cada coluna representa uma propriedade ou atributo do objeto. Dessa forma, o modelo relacional permite uma visualização clara e organizada dos dados.

No aspecto de integridade, o modelo relacional impõe certas restrições que garantem a qualidade dos dados. Isso inclui a restrição de chave primária, que garante que cada linha da tabela seja única, e a restrição de chave estrangeira, que mantém a integridade referencial entre diferentes tabelas.

Por fim, no aspecto de manipulação, o modelo relacional oferece uma série de operadores para que seja possível manipular as tabelas, como o *SELECT*, o *INSERT*, o *UPDATE* e o *DELETE*. Esses operadores permitem que o usuário selecione, insira, atualize ou exclua dados do banco de dados, tornando-o uma ferramenta flexível e útil para gerenciar grandes volumes de informações.

Uma das principais vantagens do modelo relacional é a sua uniformidade, onde cada linha da tabela possui o mesmo formato, representando um objeto do mundo real. Isso facilita a compreensão e a visualização dos dados, tornando o processo de análise mais simples e preciso (FAROULT, 2006). A Figura 1 representa uma tabela de um banco de dados relacional fictício, onde um registro nesta tabela é composta por valores nas colunas "IDENTIFICACAO", "NOME", "IDADE", "SALARIO", sendo "IDENTIFICACAO"onde

as chaves primárias de cada registro ficarão persistidas.

IDENTIFICACAO	NOME	IDADE	SALARIO
12345	JOÃO	32	3230.00
11111	MARIA	45	6125.87
33333	JOSÉ	23	2000.00

Figura 1 – Exemplo de tabela relacional

2.1.1 SQL

A SQL(*Structured Query Language*, em português "Linguagem de Consulta Estruturada") é uma linguagem de programação que permite aos usuários interagirem com bancos de dados relacionais. Ela é uma linguagem com padronização mantido pela ISO (*International Organization for Standardization*). O padrão SQL define a sintaxe e a semântica de instruções buscando garantir a portabilidade e interoperabilidade entre diferentes sistemas de gerenciamento de bancos de dados relacionais (ISO, 2023). Então mesmo que existam pequenas diferenças entre as implementações de SQL em diferentes bancos de dados, o padrão SQL ajuda a garantir que grande parte dos comandos seja compatível.

A SQL fornece uma variedade de recursos e funcionalidades que permite que os usuários realizem tarefas de manipulação de dados em bancos de dados relacionais via comandos, sendo estes organizados em cinco subconjuntos: DDL, DQL, DML, DCL e DTL. O DDL(*Data Definition Language* - Linguagem de Definição de Dados) é o conjunto de comandos responsáveis por definir a estrutura do banco de dados. Ela permite a criação, alteração e exclusão de tabelas e outros objetos do banco de dados por comandos como CREATE, RENAME, ALTER e DROP. A Figura 2 mostra como seria um comando em SQL para criar a tabela da Figura 1.

```sql
CREATE TABLE FUNCIONARIOS (
    IDENTIFICACAO INT PRIMARY KEY,
    NOME VARCHAR(50),
    IDADE INT,
    SALARIO DECIMAL(10, 2)
);
```

Figura 2 – Exemplo de comando DDL

O DQL(*Data Query Language* - Linguagem de Consulta de Dados) possui apenas um comando, SELECT, responsável por recuperar dados de uma ou mais tabelas. O DML(*Data Manipulation Language* - Linguagem de Manipulação de Dados) é utilizado

para manipular dados armazenados em tabelas do banco de dados. Comandos como INSERT, UPDATE e DELETE permitem a inserção, atualização e exclusão de registros nas tabelas. A Figura 3 mostra como seria um comando em SQL para inserir um novo registro na mesma tabela "FUNCIONARIOS".

```
INSERT INTO FUNCIONARIOS
(IDENTIFICACAO, NOME, IDADE, SALARIO)
VALUES
(123123, 'ANA', 29, 4500.00);
```

Figura 3 – Exemplo de comando DML

Já o DCL(*Data Control Language* - Linguagem de Controle de Dados) lida com os direitos de acesso e permissões de usuários dentro do banco de dados. Comandos DCL como GRANT e REVOKE são usados para conceder ou revogar privilégios a usuários e definir as permissões de acesso para tabelas e outros objetos do banco. Por fim, o DTL(*Data Transaction Language* - Linguagem de Transação de Dados) que envolve gerenciamento e controle de transações no banco de dados. Transação é o nome dado a um conjunto de uma ou mais operações que compõem uma única tarefa ou unidade lógica de trabalho a ser executada naquele banco de dados, de forma a garantir que tudo seja executado por completo, ou caso contrário, toda transação é abortada. Comandos como BEGIN TRANSACTION, COMMIT e ROLLBACK compoem este grupo.

A Figura 4 mostra como seria um comando em SQL para manipular a tabela onde pretende-se atualizar em 10% os salários dos funcionários que possuem um salário de, no mínimo, R$3000.00, utilizando-se de alguns dos operadores citados, em uma única transação.

```
BEGIN;
UPDATE FUNCIONARIOS
SET SALARIO = SALARIO * 1.10
WHERE SALARIO >= 3000.00;
COMMIT;
```

Figura 4 – Exemplo de transação SQL

2.2 NoSQL

O NoSQL(*Not Only SQL*, em português "não apenas SQL") é um termo genérico usado para caracterizar os bancos de dados não relacionais(HARRISON, 2015). Apesar de apresentarem diferentes modelos e arquiteturas, compartilham algumas características fundamentais em comum, como, por exemplo, sua estrutura flexível: os bancos de dados NoSQL permitem o armazenamento de dados sem a necessidade de estabelecer uma estrutura predefinida, sendo assim, adequados para sistemas com registros de estruturas variáveis, e em consequência disso, permite um desenvolvimento mais ágil.

Estes bancos de dados são construidos de forma que se permite uma arquitetura distribuída, ou seja, podem ser escalados horizontalmente, aumentando a quantidade de máquinas do sistema sob demanda, sendo esta uma enorme vantagem em relação aos bancos de dados relacionais, que normalmente apenas dão suporte ao escalonamento vertical(aumento de recursos na mesma máquina), sendo uma opção mais cara. Esta capacidade de escalonamento horizontal faz com que os bancos de dados NoSQL possam ser distribuídos geograficamente, em diversos *data centers*, aumentando sua resiliencia e desempenho em diferentes localidades, além de permitir uma alta disponibilidade, minimizando recursos inativos e se recuperando de falhas de forma rápida.

O NoSQL possui quatro principais tipos(BANHUDO, 2020), com diferentes tecnologias buscando resolver diversos problemas de *Big Data* que não seriam possíveis em bancos de dados relacionais. Cada um desses modelos tem suas próprias características e benefícios, e é utilizado para resolver diferentes tipos de problemas de armazenamento e processamento de dados em larga escala. Estes modelos foram brevemente descritos nas seções seguintes.

2.2.1 Chave-valor

Como o próprio nome diz, possui um modelo simples de armazenamento, onde é possível salvar um valor para uma chave específica, ou buscar e deletar valores a partir desta chave. No modelo chave-valor, os dados são organizados de forma que cada registro seja associado a uma chave única, e esse par chave-valor é armazenado no banco de dados. Essa estrutura permite um acesso rápido aos dados, uma vez que a busca por uma chave específica é altamente eficiente.

Redis é um exemplo de banco com armazenamento em chave-valor, onde os dados são mantidos em memória, melhorando drasticamente a performance de leituras e escritas no banco de dados (REDIS, 2022). Ao utilizar o Redis, é possível armazenar informações como caches, contadores, sessões de usuários e outras estruturas de dados que precisam de acesso rápido e baixa latência.

A simplicidade do modelo chave-valor torna esse tipo de banco de dados adequado

para diversos cenários, especialmente quando a velocidade e a escalabilidade são requisitos críticos. Uma visualização deste modelo é apresentada na Figura 5, onde as chaves são "nome", "idade"e "salario", e seus valores são "João", "32"e "3200"respectivamente.

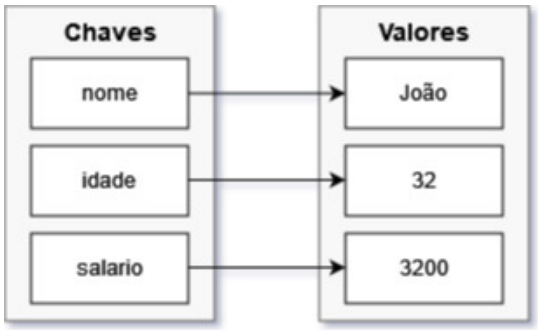

Figura 5 – Exemplo de armazenamento baseado em chave-valor.

2.2.2 Colunas

Os bancos de dados colunares organizam os dados por colunas, no lugar de linhas (que é como os bancos de dados relacionais normalmente armazenam os dados). Isso significa que todas as informações em uma coluna específica (como "idade"em uma tabela de "funcionarios", por exemplo) são armazenadas juntas. Isso pode ser particularmente útil quando você precisa fazer muitas consultas que envolvem apenas um pequeno número de colunas da tabela, porque o banco de dados só precisa ler os dados daquelas colunas específicas.

O armazenamento orientado a colunas é um fator essencial para a performance analítica, uma vez que reduz significativamente os requisitos de E/S do disco. Esses bancos são idealmente utilizados em aplicações com grande volume de dados e aplicativos analíticos, além de serem especialmente eficientes para manusear matrizes com dados esparsos (AMAZON, 2023).

Apache Cassandra é um exemplo popular de banco de dados NoSQL colunar(APACHE, 2023), desenvolvido inicialmente pelo Facebook e posteriormente tornado um projeto de código aberto pela Apache Software Foundation. A Figura 6 representa o modelo colunar, onde todos os "nomes", por exemplo, poderiam ser resgatados sem a necessidade da busca de outras informações como "idades"e "salários".

NOMES	IDADES	SALARIOS
João	32	3200
Maria	45	6125
José	23	2000
...	...	...

Figura 6 – Armazenamento colunar.

2.2.3 Grafos

Ao invés de usar tabelas ou documentos, o modelo de dados NoSQL de grafos utiliza "nós"(ou "vértices") e "relacionamentos"(ou "arestas") para representar as informações. Os nós são entidades individuais, como pessoas, lugares ou coisas, e os relacionamentos são conexões que ligam esses nós entre si(AMAZON, 2023). Um exemplo de caso de uso muito comum deste modelo é no mapeamento das rotas de uma cidade. Os nós seriam os pontos de interesse (por exemplo parques, shoppings, terminais rodoviários) e os relacionamentos seriam as ruas ou caminhos que ligam esses pontos. O Neo4j é um exemplo popular de banco de dados NoSQL baseado no modelo de grafos, que utiliza estruturas de grafo para armazenar, gerenciar e consultar os dados. Este modelo faz com que a busca por dados relacionados seja muito mais rápida que em bancos relacionais, uma vez que os relacionamentos são pré-computados(ZHANG et al., 2022). Um exemplo de utilização de grafos está presente na Figura 7 onde é feito um mapeamento da rede viária do hipercentro de Belo Horizonte.

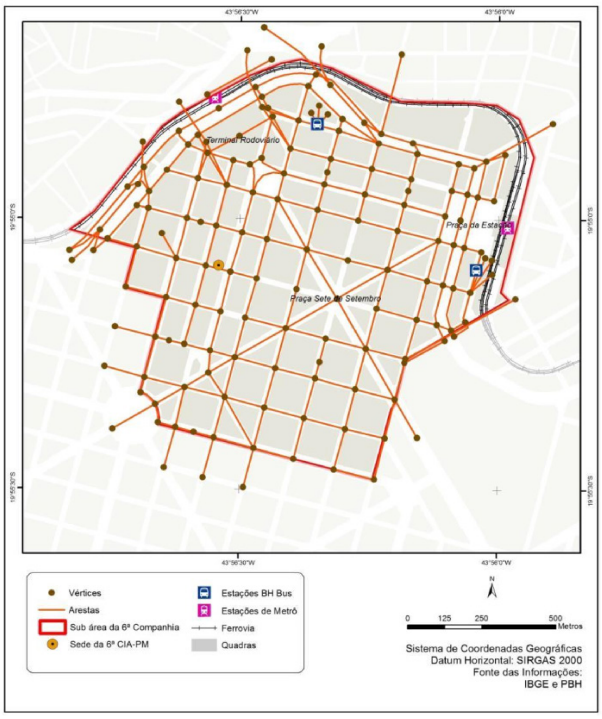

Figura 7 – Mapa contendo Grafo modelado a partir da rede viária do hipercentro de BH. Fonte: (FARIA; ALVES; BARROSO, 2019)

2.2.4 Documentos

Bancos de dados que persistem documentos são os mais comumente usados dentre os modelos NoSQL(DB-ENGINES, 2022), criado com foco em alta performance, disponibilidade e escalabilidade. Além disso, muitos bancos de dados de documentos oferecem consistência eventual, o que significa que, em sistemas distribuídos, a consistência dos dados é alcançada após algum tempo. Ele possui o mesmo conceito de seu tipo antecessor de chave-valor, porém permite trabalhar com objetos multi-camadas, chamados de documentos semiestruturados ou não estruturados. São ideais para armazenar objetos complexos e aninhados, como dados JSON, embora outros formatos como BSON (*Binary JSON*) e XML também possam ser usados, tornando-os especialmente úteis em aplicações web e mobile.

Um banco de dados de documentos é composto por coleções, que são agrupamentos de documentos, que poderiam ser comparados à tabelas do modelo relacional, porém sem um esquema de modelagem rígido. No atual contexto, um documento é uma estrutura de dados complexa que pode conter vários pares de chave-valor. Os documentos podem também conter estruturas de dados aninhadas como listas e outros documentos. Cada documento possui um identificador único em uma coleção, e da mesma forma, cada coleção também tem um nome único no banco.

Um exemplo de banco de dados de documentos é exibido na Figura 8, onde a coleção "funcionarios"exibe documentos com atributos ausentes, e a coleção "enderecos"que mostra um documento com uma lista aninhada de outros documentos.

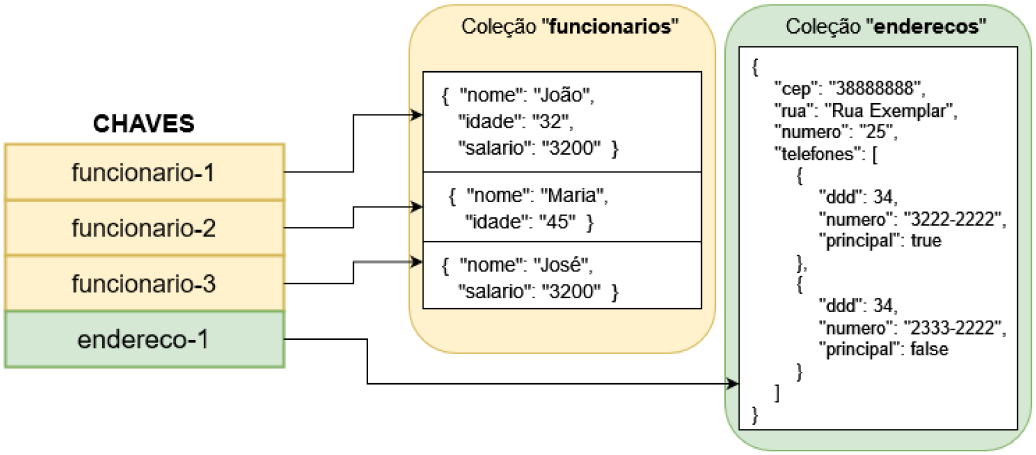

Figura 8 – Armazenamento em documentos.

Os bancos de dados de documentos suportam consultas eficientes sobre os campos dentro dos documentos, o que torna a recuperação de dados rápida e fácil. Exemplos de bancos de dados de documentos incluem MongoDB(MONGODB, 2023), CouchDB(COUCHDB, 2023) e RavenDB(RAVENDB, 2023). Essas tecnologias são ampla-

mente utilizadas em aplicações modernas, como redes sociais, aplicativos de mensagens, sistemas de gerenciamento de conteúdo, dentre outors, devido à sua capacidade de lidar com dados complexos e se adaptar facilmente a mudanças nos esquemas de dados.

As consultas em bancos de dados de documentos podem ser um pouco diferentes das consultas em bancos de dados que utilizam o SQL, pois geralmente usam um formato de consulta orientado às coleções e aos documentos e seus atributos, e variam de acordo com o sistema gerenciador do banco de dados. O MongoDB utiliza o *MongoDB Query Language (MQL)* para realizar a criação e interação com o banco de dados.

Partindo de uma base semelhante à da Figura 8, uma consulta para buscar todos os funcionários usando o MQL, a função `"find"` poderia ser utilizada. Logo, o comando seria `db.funcionarios.find()`. Em SQL, a consulta correspondente seria `SELECT * FROM funcionarios`.

Se a intenção for buscar um funcionário pelo nome, a consulta em MQL seria algo como `db.funcionarios.find({ "nome": "João"})`. Em SQL, a consulta equivalente seria `SELECT * FROM funcionarios WHERE nome = 'João'`.

Para atualizar um registro, o método `"update"` pode ser usado em MQL. Um exemplo de comando para atualizar um registro seria: `db.funcionarios.update({ "nome": "José"}, { $set: { "idade": "23"} })`. Esse comando usa o atributo `"nome"` para buscar o documento e atualiza o atributo `"idade"` do documento para o valor `"23"`. Em SQL, o comando equivalente seria `UPDATE funcionarios SET idade = '23' WHERE nome = 'José'`.

Para deletar um registro, em MQL, o comando seria `db.funcionarios.remove({ "nome": "José"})`, que remove todos os registros com o nome `"José"`. Em SQL, o comando correspondente para realizar essa ação seria `DELETE FROM funcionarios WHERE nome = 'José'`.

2.3 JSON

De acordo com a documentação do ECMA (2017), o JSON(*JavaScript Object Notation*) é uma sintaxe leve e fácil de ler e escrever para troca de dados. Ele é baseado em dois tipos de estruturas de dados: um conjunto de pares "nome/valor"e uma lista ordenada de valores. Essas estruturas podem ser aninhadas para formar estruturas de dados complexas.

Uma das principais vantagens do JSON é a sua interoperabilidade com várias linguagens de programação, tornando-o uma opção popular para a comunicação entre diferentes sistemas. Além disso, ele pode ser facilmente lido e manipulado por humanos e máquinas, o que o torna uma opção viável para o armazenamento de dados em sistemas

web e mobile.

O ECMA (2017) também destaca que o JSON é independente de plataforma e pode ser facilmente integrado em qualquer aplicação que suporte a manipulação de strings. Ele é um formato de dados amplamente utilizado na web e pode ser encontrado em diversas aplicações, desde serviços de API até sistemas de armazenamento de dados em nuvem. A Figura 9 mostra um JSON que representa uma pessoa hipotética.

Figura 9 – Exemplo de JSON.

Neste documento, as chaves e valores do tipo *string* estão encapsulados por aspas duplas. Os pares chave-valor são separados por vírgulas e podem ser agrupados em objetos, denotados por chaves "{}", ou listas, representadas por colchetes "[]". Por exemplo, a chave "endereco" refere-se a um objeto contendo informações de endereço, enquanto a chave "hobbies" refere-se a uma lista de atividades. Estas estruturas permitem a representação de dados complexos de forma organizada e hierárquica.

2.4 API

Abreviação de *Application Programming Interface*, em português "Interface de Programação de Aplicações", é um conjunto de ferramentas, definições e protocolos que possibilitam uma maneira de integrar diferentes sistemas pela rede, seja essa interna(em uma empresa) ou até mesmo pela internet. Desta forma, uma aplicação não se limita aos dados locais, ou aos bancos de dados governados por ela. (ROUSE, 2021)

3 Trabalhos Correlatos

Nesta seção, foram expostos alguns trabalhos de pesquisa condizentes ao assunto tratado neste trabalho, uma vez que diversos pesquisadores já se aventuraram em colocar em prática exercícios que buscam obter resultados que atendam diversos requisitos. Sem muita dificuldade, por meio de ferramentas de busca online em repositórios de artigos como o Google Acadêmico e *ScienceDirect* com termos relacionados a este trabalho, como "comparação sql nosql", foram encontrados dezenas de trabalhos correlatos a este. Foram selecionados alguns artigos que discorrem e comparam diferentes bancos de dados, e aqui serão expostos alguns para fins de análise e comparação.

Alguns destes trabalhos se concentram na apresentação e comparação conceitual de diferentes tipos de bancos de dados, enquanto outros apresentam um foco experimental, dando ênfase ao desempenho em testes empíricos para realizar suas comparações e conclusões a cerca dos bancos analisados. Em especial, o artigo de Deari et al. (2018) realiza algo muito próximo ao objetivo deste trabalho, comparando diretamente as características e diferenças de performance entre o MySQL e o MongoDB, a partir de diversas operações nestes bancos de dados.

Em trabalho publicado por Mohamed, Altrafi e Ismail (2014), os autores abordam conceitualmente os bancos de dados NoSQL, discutindo sua evolução e as motivações por trás do amplo uso deste tipo de banco de dados em grandes empresas, especialmente quando há a necessidade de lidar com grandes volumes de dados. Ao longo do trabalho, os autores realizaram análises das principais áreas de aplicabilidade e segurança deste tipo de banco de dados relativamente novo, comparando-o com os tradicionais bancos relacionais.

A conclusão do artigo é que a escolha entre bancos de dados relacionais e NoSQL deve ser baseada nas necessidades específicas da aplicação, considerando as vantagens e desvantagens de cada tipo de banco de dados. De acordo com os autores, os bancos de dados relacionais são adequados para aplicações que exigem uma estrutura de dados rígida e que precisam garantir consistência em todas as operações. Já os bancos de dados NoSQL são mais adequados para aplicações que precisam lidar com grandes volumes de dados distribuídos em diferentes servidores e que não precisam de uma estrutura de dados rígida.

Os autores Sahatqija et al. (2018) em artigo publicado em convenção, analisa qualitativamente os prós e contras dos bancos de dados relacionais e NoSQL, com base em materiais publicados nos últimos anos sobre o assunto. A partir dessa análise, os autores concluem que os modelos de dados não relacionais não vieram para substituir o

modelo relacional, mas sim para solucionar problemas que não eram resolvidos por ele.

A migração de bancos de dados é um processo desafiador para os desenvolvedores, e é importante considerar cuidadosamente as vantagens e desvantagens de cada modelo antes de decidir pela mudança. Os autores destacam que, embora os bancos de dados NoSQL ofereçam algumas vantagens em relação aos bancos de dados relacionais, como escalabilidade e flexibilidade, eles também apresentam desafios, como a falta de padronização e a necessidade de um conhecimento técnico mais específico para o desenvolvimento e manutenção.

O artigo de Kumar, Srividya e Mohanavalli (2017) publicado em conferência, se concentra em examinar o desempenho de dois sistemas de armazenamento NoSQL, o MongoDB e o CouchDB, por meio de uma série de operações comuns, como inserção, remoção, atualização e busca em um contexto de aplicação de *streaming*.

Os resultados da pesquisa mostram que ambos os sistemas apresentam vantagens e desvantagens em diferentes áreas, mas, em geral, o MongoDB apresentou melhor desempenho em operações de leitura e gravação, enquanto o CouchDB se saiu melhor em operações de consulta e atualização. Além disso, o CouchDB apresentou melhor escalabilidade horizontal, enquanto o MongoDB se mostrou mais eficiente em escalabilidade vertical.

Também em artigo, os autores Deari et al. (2018) fornecem uma análise comparativa e empírica entre bancos de dados baseados em documentos e bancos de dados relacionais. Eles expoem e avaliam os princípios de armazenamento e gestão de dados de cada tipo de banco de dados em questão e avaliam o desempenho das operações CRUD (Criar, Ler, Atualizar, Deletar) usando diferentes cenários no MongoDB e MySQL, que são representantes de seus respectivos modelo de bancos de dados.

Durante testes de inserção, o MongoDB mostrou-se mais eficiente, podendo inserir até 1 milhão de registros simultaneamente, enquanto o MySQL travou ao tentar inserir 100.000 registros. Em leituras individuais, ambos os sistemas tiveram desempenhos semelhantes, mas em leituras concorrentes, o MongoDB superou o MySQL. Na operação de atualização, a diferença de desempenho tornou-se mais notável com grandes volumes de dados, com ambos os sistemas enfrentando dificuldades em cenários de alta concorrência.

4 Método de Trabalho

O trabalho aborda de forma quali-quantitativa os dados para compreender o comportamento das bases de dados, a partir de repetidas execuções e monitoramento de tarefas estabelecidas, como a de provisionamento físico do banco de dados, leitura dos documentos, manipulação destes , dentre outros. Este capítulo descreve os procedimentos de ambientação, obtenção dos dados e códigos de programação necessários para realização das tarefas estipuladas nos bancos de dados.

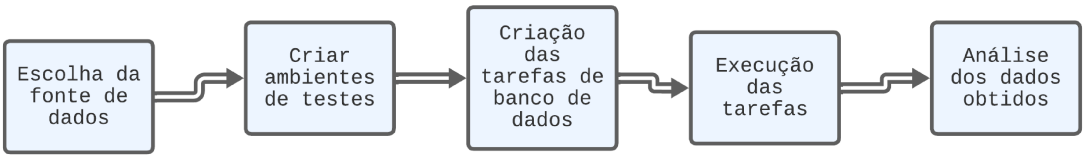

Figura 10 – Fluxo do método de trabalho.

A Figura 10 ilustra o as etapas de desenvolvimento deste trabalho, que inclui desde elencar uma fonte de dados para o uso nas tarefas de banco, à criação do ambiente e códigos necessários para extração dos dados, e posteriormente a análise dos mesmos.

4.1 Coleta dos dados

Como fonte de dados, utiliza-se da API(exposto na Seção 2.4) pública de localidades do IBGE (2021) como fonte de dados, optando pela busca de distritos brasileiros. Esta escolha é justificada pela quantidade e qualidade dos dados fornecidos já no formato JSON. Através desta API se faz possível acessar de forma rápida uma lista precisa e atualizada com mais de 10.670 distritos brasileiros. Além disso é uma API mantida pelo IBGE, então além de se garantir dados atualizados, poderá ser utilizada em uma possível reprodução desta etapa do trabalho por futuros pesquisadores. Um exemplo de documento JSON advinda desta API presente na Figura 11.

Figura 11 – Exemplo de objeto de retorno da API de localidades IBGE.

4.1.1 Integração com a API

Para realizar a busca pelos documentos JSON é necessário criar, neste caso, um script que se integre ao serviço de distritos na API de localidades do IBGE. Foi então desenvolvido um script em Python, com o auxílio das bibliotecas "requests" e "json", que se conecta e busca a lista completa de distritos brasileiros, sendo a biblioteca "requests" responsável pelo client http da transação, e a "json" para armazenar em um tipo de dado da linguagem que facilitará o uso da lista de distritos posteriormente nas execuções das tarefas de banco de dados. Um recorte do código construído está presente na Figura 12.

```python
import requests
import json

# busca dados da API de distritos do IBGE
url = "https://servicodados.ibge.gov.br/api/v1/localidades/distritos"
response = requests.get(url)
json_list = response.json()
```

Figura 12 – Trecho de código de integração.

4.2 Ambientação

A instalação dos serviços e execução das tarefas de banco de dados foram executados em um computador pessoal, possuindo este, sistema operacional Debian 10.13, com um processador AMD Ryzen 5 3600XT de seis núcleos, 32 GB de memória RAM DDR4 e um disco rígido de 1 TB. Dado a necessidade de se obter um cenário estável e coerente para tais tarefas, foram usados containers Docker para virtualização dos ambientes necessários.

4.2.1 Ambiente MySQL

Para provisionar o ambiente MySQL, usaremos a imagem Docker oficial MySQL. Ela é recuperada automaticamente do Docker Hub e criado um *container* com o comando na Figura 13.

```
docker run -d \
  --name mysql-container \
  -e MYSQL_ALLOW_EMPTY_PASSWORD=yes \
  -e MYSQL_USER=usuario \
  -e MYSQL_DATABASE=mysqldb \
  --cpus=1 \
  --memory=2g \
  -p 3306:3306 \
  -v /home/thiagop/projetos/tcc/mysql/init.sql:/docker-entrypoint-initdb.d/init.sql \
  mysql:5.7
```

Figura 13 – Comando de instalação MySQL.

Este comando cria e executa um novo *container* Docker utilizando a imagem MySQL na versão 5.7, que é a última versão estável no momento que esta atividade está sendo executada. Ao executar este comando, as seguintes ações são realizadas:

1. O *container* opera em modo *detached* (`run -d`), o que permite sua execução em segundo plano sem exibir os logs no terminal.
2. Ele é renomeado para `mysql-container` utilizando o argumento `-name mysql-container`.
3. Um usuário padrão `root` é criado com autenticação desabilitada, o que significa que não possui senha definida (`-e MYSQL_ALLOW_EMPTY_PASSWORD=yes`).
4. Adicionalmente, cria-se um usuário `usuario` com acesso ao banco `mysqldb` usando os parâmetros `-e MYSQL_USER=usuario -e MYSQL_DATABASE=mysqldb`.
5. O recurso do *container* é restrito a 1 CPU (`-cpus=1`) e 2GB de memória (`-memory=2g`).

6. A porta 3306 do *container* é mapeada para a mesma porta na máquina hospedeira, usando `-p 3306:3306`.

7. Por fim, o arquivo `init.sql` é copiado para o *container* e é executado assim que o serviço for iniciado.

```
CREATE USER 'usuario'@'%' IDENTIFIED BY '';
GRANT ALL PRIVILEGES ON *.* TO 'usuario'@'%';
FLUSH PRIVILEGES;

CREATE TABLE DISTRITOS (
    id INT PRIMARY KEY AUTO_INCREMENT,
    distrito JSON
);
```

Figura 14 – Arquivo de inicialização MySQL.

O arquivo de inicialização, quando executado, é responsável por dar as permissões necessárias para o usuário de aplicação, que posteriormente executará os comandos SQL no banco de dados. Ele também cria uma tabela `DISTRITOS`, com duas colunas: `id`, uma chave primária do tipo inteiro que é incrementada automaticamente para cada nova linha adicionada à tabela (devido à chave `AUTO_INCREMENT`), e a coluna `distrito` do tipo JSON, que armazenará dados no formato documento JSON.

4.2.2 Ambiente MongoDB

O ambiente do MongoDB, assim como o do MySQL, é criado via Docker com o comando mostrado na Figura 15.

```
docker run -d \
  --name mongodb-container \
  -p 27017:27017 \
  --cpus=1 \
  --memory=2g \
  mongo:6.0.5
```

Figura 15 – Comando de instalação MongoDB.

Ele cria e executa um *container* Docker usando a imagem oficial do MongoDB na versão 6.0.5, com as seguintes configurações:

1. O *container* opera em modo *detached* (`run -d`), permitindo sua execução em segundo plano sem exibir os logs no terminal.
2. O *container* é renomeado para `mongodb-container` usando o argumento `-name mongodb-container`.
3. A porta 27017 do *container* é mapeada para a porta 27017 da máquina hospedeira, utilizando `-p 27017:27017`.
4. O *container* tem seu uso restrito a 1 CPU (`-cpus=1`) e 2GB de memória (`-memory=2g`).

Não é necessário um script de inicialização ou configuração extra, pois a própria aplicação Python criará um novo banco de dados e as coleções em seus scripts de inserção.

4.3 Execução e Mensuração das Tarefas

Neste capítulo é apresentada uma descrição detalhada dos códigos de programação desenvolvidos para a execução das tarefas, abrangendo tanto a implementação das rotinas quanto a realização da medição do tempo médio de cada uma delas. No total, foi criado um conjunto de seis scripts Python para realizar os testes comparativos, sendo estes de inserção, busca e atualização de dados para MongoDB e MySQL, de forma separada, porém, eles possuem alguns aspectos em comum.

Primeiramente, estes scripts compartilham da mesma fonte de dados, descrita na seção 4.1, e compartilham da mesma implementação para fazer esta integração de busca dos documentos JSON pela API de Localidades do IBGE. Além disso, todos os scripts possuem a mesma definição do valor constante para o atributo chamado "qtd_execucoes", que indica o número de execuções que foram realizadas para cada teste, atribuída ao valor inteiro dez(10), ou seja, cada teste para uma operação de dados foi executada dez vezes. O mesmo vale para o atributo em lista de inteiros "qtd_registros_tarefa", que contém as quantidades de registros a serem manipulados em cada teste, também de forma consistente em todas implementações, possuindo uma lista com os valores 10, 100, 1000, 10000. Em resumo, toda rotina foi executada 10 vezes, com uma quantidade de atributos variando entre 10, 100, 1000 e 10000 valores por rotina. A Figura 16 mostra como, de fato, estão declarados os atributos citados acima, nos scripts Python:

```
# quantidade de execuções e registros por teste
qtd_execucoes = 10
qtd_registros_tarefa = [10, 100, 1000, 10000]
```

Figura 16 – Declaração de atributos estáticos

Também há semelhanças na forma que o tempo médio de execução das tarefas é capturado e manipulado. Em todos scripts, a mensuração é feita pelo módulo "time"da linguagem Python, então este módulo é importado no início de todo arquivo, junto a outros módulos auxiliares. Todo script segue uma estrutura semelhante à Figura 17.

```
import time

tempo_total = 0
for i in range(qtd_execucoes):
        start_time = time.time()
        [OPERAÇÕES NO BANCO DE DADO]
        end_time = time.time()
        tempo_execucao = end_time - start_time
        tempo_total += tempo_execucao
tempo_medio = tempo_total / qtd_execucoes
print(f"Tempo médio para inserir/manipular {qtd_registros} registros: {tempo_medio} segundos")
```

Figura 17 – Estrutura básica dos Scripts

No início das operações de inserção ou manipulação nos bancos de dados, a função "time.time()"é invocada para obter o tempo de início da execução, que é atribuído à variável "start_time", e o mesmo é feito ao fim de cada iteração, desta vez, na variável "end_time". Então, o tempo total é medido subtraindo o tempo final pelo inicial, armazenando-o na varíável "tempo_execucao", junto a isso, o tempo total daquela iteração é incrementado na variável "tempo_total", que ao final de todas iterações, é dividido pela quantidade de execuções ali performada, no caso deste trabalho um total de dez execuções, obtendo então o tempo médio das rotinas, guardado na variável "tempo_medio"que por sua vez é exibida no console via método "print".

4.3.1 MySQL

São descritos nessa seção os scripts utilizados para executar as tarefas no banco de dados MySQL, sendo elas inserção, busca e atualização, executadas repetidas vezes considerando diferentes quantidades de registros. Para nos comunicarmos com o banco de dados instanciado no momento da execução via script, utilizamos o módulo mysql.connector que é um conector oficial do MySQL para Python. Ele fornece uma interface para se conectar a um banco de dados MySQL e executar operações nele. A Figura 18 mostra o trecho onde é realizada a conexão com o banco de dados, disponibilizando um cursor para realização das operações em SQL, sendo este trecho de código comum entre todos scripts para as tarefas do MySQL.

```python
import mysql.connector


def insere_registros(registros, qtd_execucoes):
    # conecta ao banco de dados mysql
    cnx = mysql.connector.connect(
        host='localhost',
        port='3306',
        user='usuario',
        database='mysqldb'
    )
    cursor = cnx.cursor()
```

Figura 18 – Trecho para conexão com o MySQL

4.3.1.1 Inserção de dados

A Figura 19 exibe o trecho principal da tarefa de inserção de dados no MySQL.

```python
def insere_registros(registros, qtd_execucoes):
    tempo_total = 0
    for i in range(qtd_execucoes):
        try:
            start_time = time.time()
            for registro in registros:
                registro_json = json.dumps(registro)
                sql = "INSERT INTO DISTRITOS (distrito) VALUES (%s)"
                val = (registro_json,)
                cursor.execute(sql, val)
                cnx.commit()

            end_time = time.time()
            tempo_execucao = end_time - start_time
            tempo_total += tempo_execucao

            print(f"{len(registros)} registros (execução {i+1}/{qtd_execucoes}): tempo insert: {tempo_execucao} secs")

            #limpa todos os registros da tabela
            cursor.execute("DELETE FROM DISTRITOS")
            cnx.commit()

        except Exception as err:
            print(err)

    # fecha conexão com o banco de dados
    cursor.close()
    cnx.close()

    tempo_medio = tempo_total / qtd_execucoes
    print(f"Tempo médio para inserir {len(registros)} registros: {tempo_medio} segundos")
```

Figura 19 – Trecho para Inserção de Registros no MySQL

Para esta etapa do trabalho, reutilizando-se dos artifícios já discorridos, um script Python foi criado para testar o desempenho da inserção de dados no banco de dados MySQL, em coluna do tipo documento JSON. É definida uma função chamada "insere_registros"que recebe dois argumentos, "registros"(a lista de objetos JSON a serem inseridos no banco de dados) e "qtd_execucoes" (o número de vezes que o processo de inserção deve ser repetido, neste caso, 10 vezes).

A função se conecta ao banco de dados MySQL provisionado, itera em cada objeto JSON na lista fornecida pela API de localidades do IBGE e insere o objeto na tabela "DISTRITOS". Esta mesma função foi utilizada nos demais scripts de rotinas no MySQL. Por fim, a função limpa a tabela excluindo todos os registros, fecha a conexão com o banco de dados e imprime o tempo médio gasto para inserir os registros.

Em resumo, com os dados resultantes da API, ele executa a operação de inserção com uma quantidade estipulada de registros, e para cada quantidade, a operação é repetida 10 vezes, e calcula a média do tempo de execução para cada uma das quantidades.

4.3.1.2 Busca de dados

Para esta tarefa, foi criado um script Python que se conecta ao banco de dados MySQL, popula inicialmente a tabela onde foram realizadas várias pesquisas no banco de dados para medir o desempenho. A Figura 20 mostra o trecho principal deste script.

```python
def busca_registros(qtd_execucoes, qtd_registros):
    # executa as buscas e calcula o tempo médio total
    tempo_total = 0
    for i in range(qtd_execucoes):
        try:
            start_time = time.time()
            # executa a busca de registros na tabela
            sql = f"SELECT * FROM DISTRITOS LIMIT {qtd_registros}"
            cursor.execute(sql)
            result = cursor.fetchall()
            end_time = time.time()
            tempo_execucao = end_time - start_time
            tempo_total += tempo_execucao

            print(f"{len(result)} registros (execução {i+1}/{qtd_execucoes}): tempo de busca: {tempo_execucao} secs")
        except Exception as err:
            print(err)

    # fecha conexão com o banco de dados
    cursor.close()
    cnx.close()

    tempo_medio = tempo_total / qtd_execucoes
    print(f"Tempo médio para buscar {len(result)} registros: {tempo_medio} segundos")
```

Figura 20 – Trecho para Busca de Registros no MySQL

A função "busca_registros"se conecta ao mesmo banco de dados e executa uma série de pesquisas para um número especificado de registros na mesma tabela. O tempo médio gasto para executar as buscas é calculado e os resultados são impressos no console.

4.3.1.3 Atualização de dados

Para esta etapa, também foi criado um script Python que realiza a comparação do tempo de execução de tarefas de atualização de banco de dados, como exibido na Figura 21.

```python
def atualiza_registros(qtd_execucoes, qtd_registros):
    tempo_total = 0
    for i in range(qtd_execucoes):
        try:

            # busca os registros que serão modificados
            sql = f"SELECT * FROM DISTRITOS LIMIT {qtd_registros}"
            cursor.execute(sql)
            results = cursor.fetchall()

            start_time = time.time()
            for result in results:
                registro_json = result[1]

                registro = json.loads(registro_json)
                registro['nome'] = 'Novo Nome Distrito'
                registro['municipio']['nome'] = 'Novo Nome Municipio'
                registro['municipio']['microrregiao']['nome'] = 'Novo Nome Microregiao'
                registro_atualizado = json.dumps(registro)

                sql = "UPDATE DISTRITOS SET distrito = %s WHERE id = %s"
                val = (registro_atualizado, result[0])
                cursor.execute(sql, val)
                cnx.commit()

            end_time = time.time()
            tempo_execucao = end_time - start_time
            tempo_total += tempo_execucao

            print(f"{len(results)} registros (execução {i+1}/{qtd_execucoes}): tempo de atualização: {tempo_execucao} segundos")

        except Exception as err:
            print(err)

    # fecha conexão com o banco de dados
    cursor.close()
    cnx.close()

    tempo_medio = tempo_total / qtd_execucoes
    print(f"Tempo médio para atualizar {len(results)} registros: {tempo_medio} segundos")
```

Figura 21 – Trecho para Atualização de Registros no MySQL

A função "atualiza_registros"se conecta ao banco de dados, recupera uma quantidade especificada de registros da tabela "DISTRITOS", atualiza alguns atributos dos registros obtidos com novos valores e mensura o tempo de execução de cada operação de atualização. A rotina completa é realizada executando a função "atualiza_registros"várias vezes com diferentes quantidades de registros para atualizar. O tempo de execução de cada operação de atualização é mensurado, e é calculada a média de tempo ao fim do número especificado de execuções, este sendo imprimido no console.

4.3.2 MongoDB

Nessa seção são descritos os scripts utilizados para executar as tarefas no banco de dados MongoDB, sendo elas as mesmas que foram executadas no MySQL: inserção,

busca e atualização, executadas repetidas vezes considerando diferentes quantidades de registros.

Para a comunicação com o MongoDB no momento da execução dos scripts, utilizamos um outro módulo, o "pymongo". Ele é uma biblioteca Python que fornece uma interface para interagir com o MongoDB.

A Figura 22 mostra o trecho onde é realizada a criação de um cliente "MongoClient"de banco de dados, e através do método "client.db"uma conexão com o banco de dados é definida e o nome do banco de dados é personalizado aqui, como "db", e logo em seguida, é criado e selecionado uma nova coleção ("collection = db.distritos"), nomeando como "distritos". Esta instância que abstrai a coleção possui as implementações dos métodos que realiza as operações necessárias no banco de dados. Este trecho de código também é reaproveitado entre todos scripts para as tarefas no MongoDB.

```python
from pymongo import MongoClient


def insere_dados_mongodb(json_list, qtd_execucoes, tam_lista_registro):
    # conecta na base de dados
    client = MongoClient('mongodb://localhost:27017/')
    db = client.db
    collection = db.distritos
```

Figura 22 – Trecho para conexão no MongoDB

Da mesma forma que nas execuções das tarefas no MySQL, a quantidade de registros e execuções para cada operação também estão definidas nas variáveis "qtd_registros_tarefa"e "qtd_execucoes", como mostrado na Figura 16.

4.3.2.1 Inserção de dados

A Figura 23 exibe o trecho principal do script para a tarefa de inserção de dados no MongoDB.

```python
def insere_dados_mongodb(json_list, qtd_execucoes, tam_lista_registro):
    total_time = 0
    # executa inserts e calcula tempo médio
    for i in range(qtd_execucoes):
        try:
            start_time = time.time()
            for json_obj in json_list[:tam_lista_registro]:
                json_dict = json.loads(json.dumps(json_obj))
                collection.insert_one(json_dict)
            end_time = time.time()
            tempo_execucao = end_time - start_time
            tempo_total += tempo_execucao

            print(f"{tam_lista_registro} registros (execução {i+1}/{qtd_execucoes}): tempo insert: {tempo_execucao} secs")
            # limpa coleção
            collection.drop()
        except Exception as err:
            print(err)

    # fecha conexão com o banco de dados
    client.close()

    # calcula tempo médio
    media_tempo = tempo_total / qtd_execucoes
    print(f"Tempo médio para inserir {tam_lista_registro} registros: {media_tempo} segundos")
```

Figura 23 – Trecho para Inserção de Registros no MongoDB

O tarefa de inserção é definina pela função chamada "insere_dados_mongodb", ela recebe como parametros uma lista de documentos JSON "json_list", sendo esta também resultado da integração com a API de Localidades do IBGE citada na seção 4.1, recebe "qtd_execucoes"que representa a quantidade de repetições de cada rotina de inserção, como nos outros testes, atribuído a um valor de 10, e "tam_lista_registro"que diz à função quantos registros devem ser considerados e inseridos neste teste.

Ele se conecta ao banco de dados MongoDB, itera sobre a listra de documentos recebida como parâmetro e na coleção "distritos"selecionada("collection = db.distritos"), insere os documentos de forma unitária ("collection.insert_one(json_dict)"), e calcula o tempo médio necessário para finalizar a tarefa, o imprimindo no console. Ao final de cada rotina, ele apaga a coleção, para que haja uma menor interferência entre os testes. Parte deste metódo de inserção foi reaproveitado nos demais scripts de tarefas de buscas e atualizações de registros no MongoDB.

4.3.2.2 Busca de dados

Para esta tarefa, também foi criado um script que popula um banco de dados MongoDB com dados da API do IBGE utilizando de uma função parecida com a descrita anteriormente, e em seguida, realiza operações de pesquisa para avaliar o desempenho do banco de dados. A Figura 24 exibe o trecho principal do script para a tarefa de busca de dados no MongoDB.

```python
def busca_registros(qtd_execucoes, qtd_registros):
    # executa as buscas e calcula o tempo médio total
    tempo_total = 0
    for i in range(qtd_execucoes):
        try:
            start_time = time.time()

            # executa a busca de registros na coleção
            result = collection.find().limit(qtd_registros)

            end_time = time.time()
            tempo_execucao = end_time - start_time
            tempo_total += tempo_execucao

            print(f"{qtd_registros} registros (execução {i+1}/{qtd_execucoes}): tempo de busca: {tempo_execucao} secs")

        except Exception as err:
            print(err)

     # fecha a conexão com o banco de dados
    client.close()

    tempo_medio = tempo_total / qtd_execucoes
    print(f"Tempo médio para buscar {qtd_registros} registros: {tempo_medio} segundos")
```

Figura 24 – Trecho para Busca de Registros no MongoDB

A função "busca_registros"recebe dois parametros, o primeiro "qtd_execucoes", indica a quantidade de repetições de cada rotina de busca e "qtd_registros"que diz ao script quantos registros são buscados. A busca é feita na coleção por meio do método de busca "find", pertencente ao módulo "pymongo", que limita a quantidade de registros pelo submétodo "limit"("collection.find().limit(qtd_registros)"). O script então calcula o tempo médio gasto para cada operação de busca e imprime os resultados no console.

4.3.2.3 Atualização de dados

Seguindo a mesma abordagem das demais rotinas já abordadas, para esta também foi criado um script Python que realiza a tarefa de atualizar registros no banco de dados e calcular o tempo médio de algumas repetições desta tarefa. A Figura 25 mostra o trecho de código responsável por isto.

```python
def atualiza_registros(qtd_execucoes, qtd_registros):
    tempo_total = 0
    for i in range(qtd_execucoes):
        try:
            registros = collection.find().limit(qtd_registros)

            start_time = time.time()
            for registro in registros:
                registro['nome'] = 'Novo Nome Distrito'
                registro['municipio']['nome'] = 'Novo Nome Municipio'
                registro['municipio']['microrregiao']['nome'] = 'Novo Nome Microregiao'
                collection.replace_one({'_id': registro['_id']}, registro)

            end_time = time.time()
            tempo_execucao = end_time - start_time
            tempo_total += tempo_execucao

            print(f"{qtd_registros} registros (execução {i+1}/{qtd_execucoes}): tempo de atualização: {tempo_execucao} segundos")

        except Exception as err:
            print(err)

    # fecha a conexão com o banco de dados
    client.close()

    tempo_medio = tempo_total / qtd_execucoes
    print(f"Tempo médio para atualizar {qtd_registros} registros: {tempo_medio} segundos")
```

Figura 25 – Trecho para Atualização de Registros no MongoDB

Aproveitando da implementação de inserção, o banco é populado previamente, e o metodo "atualiza_registros"é responsável pela execução de toda tarefa. Em resumo, ele usa do mesmo módulo do "pymongo"para buscar uma quantidade limitada de registros para serem atualizados("collection.find().limit(qtd_registros)"), quantidade esta estabelecida pelo parametro de entrada "qtd_registros". Com a coleção já em mãos, os registros são atualizados unitariamente, através do método "replace_one", também pertencente ao modulo "pymongo". O tempo médio de todas execuções é calculado e exibido no console.

5 Resultados e Discussão

Neste capítulo, é apresentada uma análise abrangente dos dados gerados pela execução das tarefas de banco de dados. Isso inclui delinear os tratamentos e cálculos realizados, interpretando os resultados obtidos. Em todas as tabelas de resultados aqui apresentadas, os valores de tempo médio são exibidos em segundos, e são os valores já obtidos via console ao fim da execução de cada script Python.

5.1 Tarefas de banco de dados

Conforme explicado na seção 4.3, cada script responsável pela executação das baterias de tarefas de banco de dados possuem também em seu código o cálculo do tempo médio de duração e o exibe ao fim de cada execução. Estes tempos foram copiados do console e organizados em tabelas que são expostas a seguir.

Tabela 1 – Tempo médio de inserção de registros

	MySQL	MongoDB
Qtd Registros	Tempo médio(s)	Tempo médio(s)
10	2,80E-02	1,80E-02
100	0,359	6,60E-02
1000	3,685	0,576
10000	36,651	5,527

A tabela 1 mostra os resultados das tarefas de inserção de registros em ambos bancos de dados. Estes resultados obtidos pela execução dos teste indica que o banco MySQL tem tempos de inserção relativamente mais longos em comparação com o MongoDB, especialmente para grandes quantidades de dados como por exemplo 10000 registros, onde a diferença de tempo de execução da rotina ultrapassa 30 segundos. Porém, a diferença de tempo de inserção de 10 registros e 100 registros entre os bancos na prática é inferior a 300 milisegundos, oque pode ser considerado ainda aceitável dependendo da finalidade da aplicação.

Tabela 2 – Tempo médio de busca de registros

	MySQL	MongoDB
Qtd Registros	Tempo médio(s)	Tempo médio(s)
10	1,06E-03	1,09E-05
100	3,17E-03	1,10E-05
1000	2,10E-02	1,14E-05
10000	0,194	1,15E-05

A tabela 2 mostra os resultados das tarefas de busca de registros. Tanto o MySQL quanto o MongoDB demonstraram tempos de busca extremamente rápidos, independentemente do volume de dados processado. Entretanto, com base no teste que envolveu a atualização de 10.000 registros, observou-se uma diferença, ainda que sutil, nos tempos de execução. Uma diferença de menos de 200 milissegundos, embora pareça insignificante, pode sugerir que, ao lidar com volumes de registros ainda maiores, o MySQL possa vir a apresentar uma degradação no tempo de resposta. Em contrapartida, o MongoDB mostrou-se estável e consistente em todas as volumetrias analisadas.

Tabela 3 – Tempo médio de atualização de registros

	MySQL	MongoDB
Qtd Registros	Tempo médio(s)	Tempo médio(s)
10	0,023	1,17E-02
100	0,367	5,30E-02
1000	3,845	0,539
10000	58,354	5,148

A tabela 3 mostra os resultados das tarefas de atualização de registros para os bancos de dados MySQL e MongoDB. Ela evidencia diferenças em sua performance dependendo da quantidade de dados manipulados. Ao analisar os resultados, é observado que para conjuntos menores de dados, como 10 e 100 registros, a diferença de tempo entre os dois bancos é menos acentuada, sendo o MongoDB um pouco mais rápido. Entretanto, à medida que o volume de dados aumenta, a discrepância entre os tempos de atualização passa a ser mais evidente. Para 1.000 registros, o MySQL demorou aproximadamente

3,845 segundos, enquanto o MongoDB precisou de apenas 0,539 segundos. Este contraste torna-se ainda mais expressivo ao lidar com 10.000 registros, onde o MySQL precisou de 58,354 segundos, comparado aos 5,148 segundos do MongoDB. Portanto, pode-se perceber que embora ambos os bancos de dados possam ser eficientes em volumes menores de dados, o MongoDB demonstra uma capacidade superior de gerenciar e atualizar grandes quantidades de registros em um tempo consideravelmente menor do que o MySQL.

Em complemento à análise aqui realizada, é relevante apontar que os estudos de Deari et al. (2018) revelam resultados em seus testes de desempenho entre MongoDB e MySQL que reforçam alguns dos comportamentos apresentados no presente trabalho. Dentre os testes realizados pelos pesquisadores, especificamente, nas operações de inserção, o MongoDB demonstrou uma eficiência considerável, sobressaindo-se particularmente em cenários com volumes menores de dados. Esta eficácia em inserções, seja realizadas individualmente ou de maneira simultânea, ressalta a adaptabilidade do MongoDB a diferentes ambientes de trabalho, oferecendo vantagens em aplicações que demandam rápida inclusão de registros. Outras conclusões de trabalhos de autores citados no Capítulo 3 que analisaram conceitualmente bancos de dados relacionais e NoSQL, como por exemplo Mohamed, Altrafi e Ismail (2014), corroboram com o resultado das tarefas de inserção, pois afirmam também que os bancos de dados NoSQL possuem funcionalidades focadas na alta disponibilidade e desempenho.

Além disso, as operações de leitura, cruciais para a eficácia de muitas aplicações, foram destacadas no trabalho de Deari et al. (2018). Em comparações diretas, o MongoDB mostrou-se superior ao MySQL, especialmente em cenários de leitura simultânea com grandes volumes de dados. Esta característica pode ser fundamental em cenários que necessitam de acessos concorrentes e recuperações rápidas, reforçando a escolha do MongoDB em aplicações de alta demanda.

Em relação às operações de atualização, os dados apresentados por Deari et al. (2018) fornecem insights valiosos que complementam a discussão presente neste trabalho. A capacidade de executar atualizações eficientes é fundamental para manter a integridade dos dados e adaptar-se a requisitos dinâmicos nas aplicações. Conforme evidenciado pelo estudo de Deari et al. (2018), o MongoDB exibe uma capacidade notável de manter um desempenho consistente, mesmo com o aumento no volume de registros a serem atualizados. Em contraste, enquanto o MySQL se mostra competente em tarefas de atualização, ele pode mostrar limitações em cenários com alta concorrência ou grandes conjuntos de dados. Essa caracterização distintiva entre MongoDB e MySQL é crucial, pois pode guiar decisões estratégicas no design e na implementação de sistemas que dependem de operações de atualização frequentes e de grande volume.

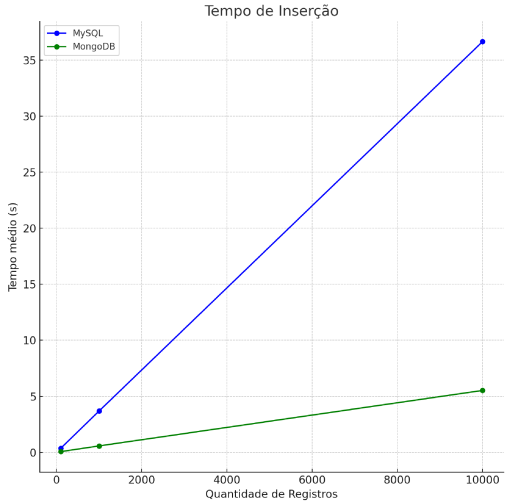

Figura 26 – Tempo médio de inserção comparando MySQL e MongoDB.

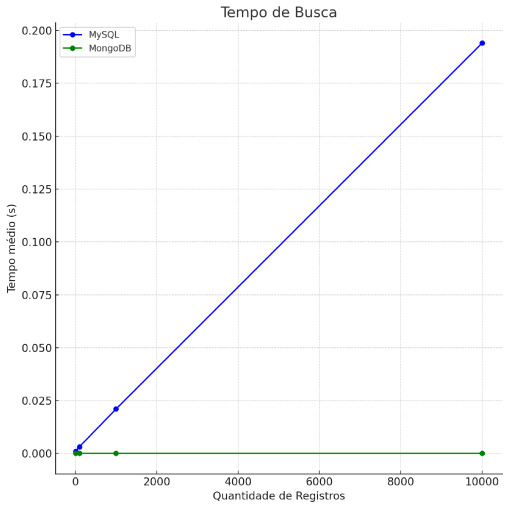

Figura 27 – Tempo médio de busca em função da quantidade de registros para MySQL e MongoDB.

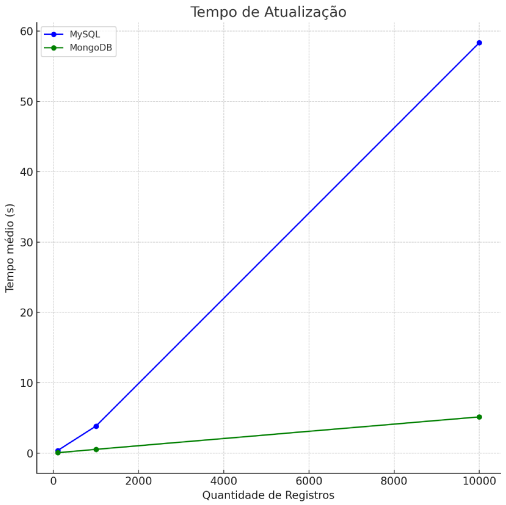

Figura 28 – Tempo médio de atualização para MySQL e MongoDB.

No contexto do estudo, as Figuras 26, 27 e 28 visualizam os dados comparativos, previamente apresentados em tabelas, sobre o desempenho dos sistemas de gerenciamento de banco de dados MySQL e MongoDB. A Figura 26 detalha as diferenças no tempo de inserção de dados, mostrando o aumento progressivo do tempo para o MySQL em contraste com a estabilidade do MongoDB. A Figura 27 concentra-se no tempo de busca, destacando a eficiência constante do MongoDB, independentemente do volume de dados, em oposição ao aumento linear observado no MySQL. Por fim, a Figura 28 compara o tempo de atualização, evidenciando a maior eficácia do MongoDB, que mantém tempos menores e mais consistentes mesmo com um grande volume de dados, sugerindo uma melhor performance e escalabilidade em comparação ao MySQL.

6 Conclusões e Recomendações para Trabalhos Futuros

O cenário atual, marcado pela Quarta Revolução Industrial, trouxe à tona a necessidade de gestão de um volume crescente de dados. Neste contexto, o MongoDB, um banco de dados NoSQL orientado a documentos, mostrou-se mais eficiente em operações específicas com documentos JSON quando comparado ao MySQL, um banco de dados relacional tradicional. Este estudo revelou que, para tarefas como inserção, busca e atualização, o MongoDB teve desempenho superior em cenários de maior volume de documentos, mas não descarta o uso da implementação do tipo JSON no MySQL em cenários de menor carga.

No entanto, é crucial ressaltar que os resultados são específicos para o contexto deste trabalho, e uma análise mais ampla é necessária antes de generalizar essas conclusões. A escolha de um sistema de banco de dados deve considerar múltiplos fatores além do desempenho, como segurança, escalabilidade e integração com outros sistemas.

Para trabalhos futuros, sugere-se a expansão da pesquisa para outros sistemas de banco de dados NoSQL, a fim de obter uma compreensão mais abrangente de seu desempenho em diferentes cenários. Além disso, a realização de estudos de caso em ambientes reais pode oferecer insights valiosos sobre desafios práticos enfrentados pelas organizações ao implementar estas tecnologias.

O código dos scripts criados e utilizados neste trabalho, em Apêndices, possui propositalmente repetições de funções chaves para os testes, como conexão nos bancos de dados e consumo de uma coleção de documentos por um cliente Http. Este código pode ser reaproveitado na construção de um framework para automação de testes em diversos outros bancos em cenários diferentes.

Em suma, enquanto este estudo forneceu uma compreensão inicial da eficácia relativa do MySQL e MongoDB na manipulação de JSON, há um vasto campo de investigação a ser explorado, e futuras pesquisas podem lançar luz sobre nuances ainda não contempladas.

Referências

AMAZON. **O que é um banco de dados em colunas? Banco de dados colunar vs. banco de dados relacional**. 2023. <https://aws.amazon.com/pt/nosql/columnar/>. Citado 2 vezes nas páginas 14 e 15.

APACHE. **Cassandra Documentation**. 2023. <https://cassandra.apache.org/doc/latest/>. Citado na página 14.

BANHUDO, G. **Exploring the NoSQL Family (A (long) primer on a growing requirement for Data Scientists)**. 2020. Disponível em: <https://pub.towardsai.net/exploring-the-nosql-family-49e9f23313ad>. Acesso em: 2022-06-10. Citado na página 13.

COUCHDB, A. **Technical Overview**. 2023. Apache CouchDB Online documentation. Disponível em: <https://docs.couchdb.org/en/stable/intro/overview.html>. Acesso em: 2023-05-15. Citado na página 16.

DATE, C. J. **Introdução a Sistemas de Bancos de Dados**. [S.l.]: Elsevier, 2004. v. 8. Citado na página 10.

DB-ENGINES. **DB-Engines Ranking**. 2022. Disponível em: <https://db-engines.com/en/ranking>. Acesso em: 2022-11-01. Citado 2 vezes nas páginas 8 e 16.

DEARI, R.; ZENUNI, X.; AJDARI, J.; ISMAILI, F.; RAUFI, B. Analysis and comparison of document-based databases with sql relational databases: Mongodb vs mysql. **Proceedings of the International Conference on Information Technologies (InfoTech-2018)**, 2018. Citado 3 vezes nas páginas 19, 20 e 36.

ECMA. **The JSON data interchange syntax**. 2017. Disponível em: <https://www.ecma-international.org/publications-and-standards/standards/ecma-404/>. Acesso em: 2022-07-06. Citado 2 vezes nas páginas 17 e 18.

FARIA, A. H. P. d.; ALVES, D. F. C.; BARROSO, L. C. Aplicação da teoria de grafos e análise espacial para solução de problemas geográficos: Um estudo da criminalidade violenta no hipocentro de belo horizonte. In: _____. [S.l.: s.n.], 2019. cap. 5, p. 65. Citado na página 15.

FAROULT, S. **The Art of SQL**. [S.l.]: O'Reilly Media, 2006. Citado na página 10.

FOOTE, K. D. **A Brief History of Non-Relational Databases**. 2018. Disponível em: <https://www.dataversity.net/a-brief-history-of-non-relational-databases>. Acesso em: 2022-06-19. Citado na página 7.

FREDERICK, D. E. **Libraries, data and the fourth industrial revolution: Data Deluge Column**. [S.l.]: Library Hi Tech News, 2016. v. 33. Citado na página 7.

HARRISON, G. **Next Generation Databases: NoSQL and Big Data**. 1. ed. [S.l.]: Apress, 2015. ISBN 978-1484213308. Citado na página 13.

IBGE. **API de Localidades**. 2021. Disponível em: <https://servicodados.ibge.gov.br/api/docs/localidades>. Acesso em: 2023-04-01. Citado na página 21.

ISO. **Information technology — Database languages SQL — Part 1: Framework**. Geneva, CH, 2023. v. 2023. Citado na página 11.

KUMAR, K. B. S.; SRIVIDYA; MOHANAVALLI, S. A performance comparison of document oriented nosql databases. **IEEE International Conference on Computer, Communication, and Signal Processing**, 2017. Citado na página 20.

MOHAMED, M. A.; ALTRAFI, O. G.; ISMAIL, M. O. Relational vs. nosql databases: A survey. **International Journal of Computer and Information Technology**, 2014. Citado 2 vezes nas páginas 19 e 36.

MONGODB. **MongoDB Documentation**. 2023. Online documentation. Disponível em: <https://docs.mongodb.com/>. Acesso em: 2023-03-01. Citado 2 vezes nas páginas 8 e 16.

ORACLE. **The JSON Data Type**. 2022. Disponível em: <https://dev.mysql.com/doc/refman/8.0/en/json.html>. Acesso em: 2022-05-01. Citado na página 7.

PHIRI, H.; KUNDA, D. Comparative study of nosql and relational database. **Zambia ICT Journal**, v. 1, 12 2017. Citado na página 7.

RAVENDB. **NoSQL Database Documentation**. 2023. RavenDB Documentation. Disponível em: <https://ravendb.net/why-ravendb>. Acesso em: 2023-05-15. Citado na página 16.

REDIS. **Introduction to Redis**. 2022. Disponível em: <https://redis.io/docs/about/>. Acesso em: 2022-04-15. Citado na página 13.

ROUSE, M. **Application Programming Interface**. 2021. Disponível em: <https://www.techopedia.com/definition/24407/application-programming-interface-api>. Acesso em: 2023-01-01. Citado na página 18.

SADALAGE, P. J.; FOWLER, M. **NoSQL Distilled: A Brief Guide to the Emerging World of Polyglot Persistence**. 1. ed. Addison-Wesley Professional, 2012. v. 1. ISBN 0321826620,9780321826626. Disponível em: <http://gen.lib.rus.ec/book/index.php?md5=6c594096c6e33ae41a6d365f2c4588c6>. Citado na página 7.

SAHATQIJA, K.; AJDARI, J.; ZENUNI, X.; RAUFI, B. Comparison between relational and nosql databases. **41st International Convention on Information and Communication Technology, Electronics and Microelectronics (MIPRO)**, 2018. Citado na página 19.

UYANGA, S.; MUNKHTSETSEG, N.; BATBAYAR, S.; BAT-ULZII, S. A comparative study of nosql and relational database. **Advances in Intelligent Information Hiding and Multimedia Signal Processing**, Springer, 2021. Disponível em: <https://link.springer.com/chapter/10.1007/978-981-33-6757-9_16>. Citado na página 7.

ZHANG, L.; LI, Z.; REN, H.; YU, X.; MA, Y.; ZHANG, Q. Knowledge graph and behavior portrait of intelligent attack against path planning. 2022. Disponível em: <https://doi.org/10.1002/int.22874>. Citado na página 15.

Apêndices

Apêndice 1 – Código usado na rotina de inserção de documentos no MySQL

```
import requests
import json
import time
import mysql.connector


def insere_registros(registros, qtd_execucoes):
    # conecta ao banco de dados mysql
    cnx = mysql.connector.connect(
        host='localhost',
        port='3306',
        user='usuario',
        database='mysqldb'
    )
    cursor = cnx.cursor()

    \# executa os inserts e calcula o tempo medio total
    tempo_total = 0
    for i in range(qtd_execucoes):
        try:
            start_time = time.time()
            for registro in registros:
                registro_json = json.dumps(registro)
                sql = "INSERT INTO DISTRITOS (distrito) VALUES (%s)"
                val = (registro_json,)
                cursor.execute(sql, val)
                cnx.commit()

            end_time = time.time()
            tempo_execucao = end_time - start_time
            tempo_total += tempo_execucao

            print(f"{len(registros)} registros (execucao {i+1}/{qtd_execucoes}):
tempo insert: {tempo_execucao} secs")

            # limpa todos os registros da tabela
            cursor.execute("DELETE FROM DISTRITOS")
            cnx.commit()

        except Exception as err:
```

```
            print(err)

    # fecha conexao com o banco de dados
    cursor.close()
    cnx.close()

    tempo_medio = tempo_total / qtd_execucoes
    print(f"Tempo medio para inserir {len(registros)} registros: {tempo_medio} segundos")


# quantidade de execucoes e registros por teste
qtd_execucoes = 100
qtd_registros_tarefa = [10, 100, 1000, 10000]

# busca dados da API de distritos do IBGE
url = "https://servicodados.ibge.gov.br/api/v1/localidades/distritos"
response = requests.get(url)
json_list = response.json()

# insere os registros para cada quantidade de registros na lista qtd_registros_tarefa
for qtd_registros in qtd_registros_tarefa:
    registros = json_list[:qtd_registros]
    insere_registros(registros, qtd_execucoes)
```

Apêndice 2 – Código usado na rotina de busca de documentos no MySQL

```
import requests
import json
import time
import mysql.connector


def popula_banco_dados():
    # conecta ao banco de dados mysql
    cnx = mysql.connector.connect(
        host='localhost',
        port='3306',
        user='usuario',
        database='mysqldb'
    )
    cursor = cnx.cursor()

    # busca dados da API de distritos do IBGE
```

```
    url = "https://servicodados.ibge.gov.br/api/v1/localidades/distritos"
    response = requests.get(url)
    json_list = response.json()

    try:
        # insere os registros na tabela de distritos
        for registro in json_list:
            registro_json = json.dumps(registro)
            sql = "INSERT INTO DISTRITOS (distrito) VALUES (%s)"
            val = (registro_json,)
            cursor.execute(sql, val)
            cnx.commit()
    except Exception as err:
        print(err)
    # fecha conexao com o banco de dados
    cursor.close()
    cnx.close()


def busca_registros(qtd_execucoes, qtd_registros):
    # conecta ao banco de dados mysql
    cnx = mysql.connector.connect(
        host='localhost',
        port='3306',
        user='usuario',
        database='mysqldb'
    )
    cursor = cnx.cursor()

    # executa as buscas e calcula o tempo medio total
    tempo_total = 0
    for i in range(qtd_execucoes):
        try:
            start_time = time.time()
            # executa a busca de registros na tabela
            sql = f"SELECT * FROM DISTRITOS LIMIT {qtd_registros}"
            cursor.execute(sql)
            result = cursor.fetchall()
            end_time = time.time()
            tempo_execucao = end_time - start_time
            tempo_total += tempo_execucao
```

```
            print(f"{len(result)} registros (execucao {i+1}/{qtd_execucoes}):
tempo de busca: {tempo_execucao} secs")
        except Exception as err:
            print(err)

    # fecha conexao com o banco de dados
    cursor.close()
    cnx.close()

    tempo_medio = tempo_total / qtd_execucoes
    print(f"Tempo medio para buscar {len(result)} registros:
{tempo_medio} segundos")


# quantidade de execucoes e registros por teste
qtd_execucoes = 100
qtd_registros_tarefa = [10, 100, 1000, 10000]

# popula a tabela de distritos com os dados da API do IBGE
popula_banco_dados()

# busca registros para cada quantidade de registros na lista qtd_registros_tarefa
for qtd_registros in qtd_registros_tarefa:
    busca_registros(qtd_execucoes, qtd_registros)
```

Apêndice 3 – Código usado na rotina de atualização de documentos no MySQL

```
import requests
import json
import time
import mysql.connector


def popula_banco_dados():
    # conecta ao banco de dados mysql
    cnx = mysql.connector.connect(
        host='localhost',
        port='3306',
        user='usuario',
        database='mysqldb'
    )
    cursor = cnx.cursor()
```

```
    # busca dados da API de distritos do IBGE
    url = "https://servicodados.ibge.gov.br/api/v1/localidades/distritos"
    response = requests.get(url)
    json_list = response.json()

    #limpa todos os registros da tabela
    cursor.execute("DELETE FROM DISTRITOS")
    cnx.commit()
    try:
        # insere os registros na tabela de distritos
        for registro in json_list:
            registro_json = json.dumps(registro)
            sql = "INSERT INTO DISTRITOS (distrito) VALUES (%s)"
            val = (registro_json,)
            cursor.execute(sql, val)
            cnx.commit()
    except Exception as err:
        print(err)
    # fecha conexao com o banco de dados
    cursor.close()
    cnx.close()


def atualiza_registros(qtd_execucoes, qtd_registros):
    # conecta ao banco de dados mysql
    cnx = mysql.connector.connect(
        host='localhost',
        port='3306',
        user='usuario',
        database='mysqldb'
    )
    cursor = cnx.cursor()

    tempo_total = 0
    for i in range(qtd_execucoes):
        try:

            # busca os registros que serao modificados
            sql = f"SELECT * FROM DISTRITOS LIMIT {qtd_registros}"
            cursor.execute(sql)
            results = cursor.fetchall()
```

```
            start_time = time.time()
            for result in results:
                registro_json = result[1]

                registro = json.loads(registro_json)
                registro['nome'] = 'Novo Nome Distrito'
                registro['municipio']['nome'] = 'Novo Nome Municipio'
                registro['municipio']['microrregiao']['nome'] = 'Novo Nome Microregiao'
                registro_atualizado = json.dumps(registro)

                sql = "UPDATE DISTRITOS SET distrito = %s WHERE id = %s"
                val = (registro_atualizado, result[0])
                cursor.execute(sql, val)
                cnx.commit()

            end_time = time.time()
            tempo_execucao = end_time - start_time
            tempo_total += tempo_execucao

            print(f"{len(results)} registros (execucao {i+1}/{qtd_execucoes}):
 tempo de atualizacao: {tempo_execucao} segundos")

        except Exception as err:
            print(err)

    # fecha conexao com o banco de dados
    cursor.close()
    cnx.close()

    tempo_medio = tempo_total / qtd_execucoes
    print(f"Tempo medio para atualizar {len(results)} registros:
{tempo_medio} segundos")


# quantidade de execucoes e registros por teste
qtd_execucoes = 100
qtd_registros_tarefa = [10, 100, 1000, 10000]

# popula a tabela de distritos com os dados da API do IBGE
popula_banco_dados()

# atualiza registros para cada quantidade de registros na lista qtd_registros_tarefa
```

```
for qtd_registros in qtd_registros_tarefa:
    atualiza_registros(qtd_execucoes, qtd_registros)
```

Apêndice 4 – Código usado na rotina de inserção de documentos no MongoDB

```
import requests
import json
import time
from pymongo import MongoClient


def insere_dados_mongodb(json_list, qtd_execucoes, tam_lista_registro):
    # conecta na base de dados
    client = MongoClient('mongodb://localhost:27017/')
    db = client.db
    collection = db.distritos

    total_time = 0
    # executa inserts e calcula tempo medio
    for i in range(qtd_execucoes):
        try:
            start_time = time.time()
            for json_obj in json_list[:tam_lista_registro]:
                json_dict = json.loads(json.dumps(json_obj))
                collection.insert_one(json_dict)
            end_time = time.time()
            tempo_execucao = end_time - start_time
            tempo_total += tempo_execucao

            print(f"{tam_lista_registro} registros (execucao {i+1}/{qtd_execucoes}):
tempo insert: {tempo_execucao} secs")
            # limpa colecao
            collection.drop()
        except Exception as err:
            print(err)

    # fecha conexao com o banco de dados
    client.close()

    # calcula tempo medio
    media_tempo = tempo_total / qtd_execucoes
    print(f"Tempo medio para inserir {tam_lista_registro} registros:
{media_tempo} segundos")
```

```
# quantidade de execucoes e registros por teste
qtd_execucoes = 100
qtd_registros_tarefa = [10, 100, 1000, 10000]

# busca dados da API de distritos do IBGE
url = "https://servicodados.ibge.gov.br/api/v1/localidades/distritos"
response = requests.get(url)
json_list = response.json()

for tam_lista_registro in qtd_registros_tarefa:
    insere_dados_mongodb(json_list, qtd_execucoes, tam_lista_registro)
```

Apêndice 5 – Código usado na rotina de busca de documentos no MongoDB

```
import requests
import json
import time
from pymongo import MongoClient


def popula_banco_dados():
    # conecta ao banco de dados mongodb
    client = MongoClient('mongodb://localhost:27017/')
    db = client.db
    collection = db.distritos

    # busca dados da API de distritos do IBGE
    url = "https://servicodados.ibge.gov.br/api/v1/localidades/distritos"
    response = requests.get(url)
    json_list = response.json()

    try:
        # insere os registros na colecao de distritos
        for registro in json_list:
            collection.insert_one(registro)
    except Exception as err:
        print(err)

     # fecha a conexao com o banco de dados
    client.close()
```

```
def busca_registros(qtd_execucoes, qtd_registros):
    # conecta ao banco de dados mongodb
    client = MongoClient('mongodb://localhost:27017/')
    db = client.db
    collection = db.distritos

    # executa as buscas e calcula o tempo medio total
    tempo_total = 0
    for i in range(qtd_execucoes):
        try:
            start_time = time.time()

            # executa a busca de registros na colecao
            result = collection.find().limit(qtd_registros)

            end_time = time.time()
            tempo_execucao = end_time - start_time
            tempo_total += tempo_execucao

            print(f"{qtd_registros} registros (execucao {i+1}/{qtd_execucoes}):
tempo de busca: {tempo_execucao} secs")

        except Exception as err:
            print(err)

     # fecha a conexao com o banco de dados
    client.close()

    tempo_medio = tempo_total / qtd_execucoes
    print(f"Tempo medio para buscar {qtd_registros} registros: {tempo_medio} segundos")


# quantidade de execucoes e registros por teste
qtd_execucoes = 100
qtd_registros_tarefa = [10, 100, 1000, 10000]

# popula a colecao de distritos com os dados da API do IBGE
popula_banco_dados()

# busca registros para cada quantidade de registros na lista qtd_registros_tarefa
for qtd_registros in qtd_registros_tarefa:
```

```
    busca_registros(qtd_execucoes, qtd_registros)
```

Apêndice 6 – Código usado na rotina de atualização de documentos no MongoDB

```
import requests
import json
import time
from pymongo import MongoClient


def popula_banco_dados():
    # conecta ao banco de dados mongodb
    client = MongoClient('mongodb://localhost:27017/')
    db = client.db
    collection = db.distritos

    # busca dados da API de distritos do IBGE
    url = "https://servicodados.ibge.gov.br/api/v1/localidades/distritos"
    response = requests.get(url)
    json_list = response.json()

    try:
        # insere os registros na colecao de distritos
        for registro in json_list:
            collection.insert_one(registro)
    except Exception as err:
        print(err)

    # fecha conexao com o banco de dados
    client.close()


def atualiza_registros(qtd_execucoes, qtd_registros):
    # conecta ao banco de dados mysql
    client = MongoClient('mongodb://localhost:27017/')
    db = client.db
    collection = db.distritos

    # executa as atualizacoes e mede o tempo
    tempo_total = 0
    for i in range(qtd_execucoes):
        try:
            registros = collection.find().limit(qtd_registros)
```

```
            start_time = time.time()
            for registro in registros:
                registro['nome'] = 'Novo Nome Distrito'
                registro['municipio']['nome'] = 'Novo Nome Municipio'
                registro['municipio']['microrregiao']['nome'] = 'Novo Nome Microregiao'
                collection.replace_one({'_id': registro['_id']}, registro)

            end_time = time.time()
            tempo_execucao = end_time - start_time
            tempo_total += tempo_execucao

            print(f"{qtd_registros} registros (execucao {i+1}/{qtd_execucoes}):
tempo de atualizacao: {tempo_execucao} segundos")

        except Exception as err:
            print(err)

    # fecha a conexao com o banco de dados
    client.close()

    tempo_medio = tempo_total / qtd_execucoes
    print(f"Tempo medio para atualizar {qtd_registros} registros:
{tempo_medio} segundos")


# quantidade de execucoes e registros por teste
qtd_execucoes = 100
qtd_registros_tarefa = [10, 100, 1000, 10000]

# popula a colecao de distritos com os dados da API do IBGE
popula_banco_dados()


# atualiza registros para cada quantidade de registros na lista qtd_registros_tarefa
for qtd_registros in qtd_registros_tarefa:
    atualiza_registros(qtd_execucoes, qtd_registros)
```